Appendix U

Improving Software Economics in the Aerospace & Defense Industry

Content

Mike Devlin
Walker Royce
Rational Corporation

U.1 Editor’s Note

If you have read Chapters 1 through 15, you have traversed virtually the full range of challenges and opportunities associated with software management. These insights, when properly addressed and exploited, will help ensure that you deliver software products on the predicted schedule, at the predicted cost, with the predicted quality and performance desired by the user. On the other hand, by now you might be just a little confused. In seeking a means to bring you an effective summary, the following paper was brought to our attention. Co-written by Mr. Mike Devlin and Mr. Walker Royce of the Rational Software Corporation, this paper succinctly identifies the software engineering practice required to produce more capable defense systems in a more timely and more economic fashion. Although we cannot, and do not, recommend the Rational Software Corporation, we do strongly recommend that you consider them as a benchmark of capability against which to compare competing purveyors of software engineering technology.

U.2 Introduction

Modern aerospace and defense systems incorporate increasingly sophisticated information processing and control systems. These systems contain large amounts of complex software directly in the operational systems themselves, and in the associated development, test, logistics and support systems. This software typically must provide extensive functionality while meeting stringent requirements for safety, security, reliability, availability, and (real-time) performance. The aerospace and defense industry has long recognized that advances in software technology and process improvement are essential to the delivery of more capable systems with shorter development cycles and lower cost.

Rational Software Corporation has been intimately involved in dealing with the pragmatic successes and failures of its customer’s software engineering projects across a broad range of Aerospace, Defense, and Commercial applications for over 12 years. The purpose of this paper is to examine three interrelated issues which bear directly on the software capability of the aerospace and defense industry and summarize the current maturity of software engineering practice from Rational’s perspective. Advances in software process, improvements in acquisition policy and a continued focus on Ada need to be integrated into a complementary approach to provide breakthrough improvements in the economics of sophisticated software development.

Software Engineering in the Aerospace and Defense Industry examines the state of software engineering in the aerospace and defense industry in comparison to best commercial practice in other industries and defines the elements of a next generation software process.

The Defense Software Acquisition Process examines DoD software acquisition policy and its impact on the economics of software development in the aerospace and defense industry.

Ada and the Aerospace and Defense Industry examines DoD policy for a continued focus on Ada and its impact on the aerospace and defense industry.

The question of the Ada policy has received some recent prominence, influencing the timing and focus of this paper. While Ada is an important issue, it is inappropriate to address the issue of Ada separately from the broader issues of software engineering and acquisition policy. This paper concludes with recommendations which are based on Rational's experience in employing advanced software technologies in both commercial and defense applications to highlight the discriminating practices of successful projects.

Recommendations presents a set of suggestions for accelerating further adoption of modern techniques within the defense and aerospace industry.

U.3 Software Engineering in the Aerospace and Defense Industry

U.3.1 Re-engineering the Software Development Process

Over the last decade there has been a significant re-engineering of the software development process, replacing many of the traditional management and technical practices with radically new approaches that combine some hard lessons of experience with advances in software engineering technology. We use the terms "object-oriented software process" and "modern techniques" to encompass these new practices. While the essence of this process can be used in most software systems, it is particularly appropriate in situations which are driven by the following needs:

Accommodating change. Those situations where requirements are expected to change over the life of the software, requirements definition requires extensive user input and iteration, or where a flexible architecture is necessary to accommodate growth and change in function, technology, or performance.

Achieving software return on investment (ROI). Those situations where economic considerations require a high degree of reuse of pre-existing components and/or newly developed components within a single system or across multiple systems within a given application domain or line of business.

Value engineering. Projects where there is a need to make accurate, rapid and flexible tradeoffs between cost, schedule, functionality, quality and performance throughout the development process.

Technical or schedule risk. Those situations where schedule pressure (based on mission requirements or time-to-market considerations) or technical uncertainty (complexity, scale, concurrent engineering) require an incremental approach with early delivery of useful versions that provide a solid foundation for further evolution into more complete products over time.

In the commercial world, the combination of competitive pressures, profitability, diversity of customers, and rapidly changing technology cause many systems to have some or all of the above characteristics. In the defense industry it is budget pressures, the dynamic and diverse threat environment, the long operational lifetime of systems, and the predominance of large scale, complex applications which cause many systems to share these characteristics. The paramount need of projects which contain some or all of the above characteristics is one of management control and adaptability. Consequently, our definition of the solution focuses primarily on process with strong support from advancing technologies in languages, environments and architectural reuse.

U.3.2 The Elements of an Object-Oriented Software Process

The salient elements of an object-oriented software process include a number of interrelated software engineering practices. We have avoided the use of the terms “megaprogramming,” “spiral model,” and “next generation software process” even though there is substantial commonality between the techniques of these process frameworks and our presentation. The following themes constitute the recurring practices of successful software projects based on our pragmatic field experience drawn from many sources.

Object-oriented analysis, design, and programming. These techniques replace traditional data-driven methods and functional decomposition methods (structured analysis and design) with an integrated approach to analysis, design and implementation based on an object model.

Rapid prototyping and iterative development. These techniques replace the conventional waterfall model. While there are variations, the basic concept is that early in the development process an initial version of the system is rapidly constructed with an emphasis on addressing high risk areas, stabilizing the basic architecture, and refining the requirements (with extensive user input where possible). Development then proceeds as a series of iterations building on the core architecture until the desired level of functionality, performance and robustness is achieved. This process places emphasis on the whole system rather than just the individual parts. Through a process of continuous integration, risk is reduced early in the project, avoiding integration surprises late in the project.

Architecture-driven development. Traditionally, the software development process has been requirements-driven, where an attempt is made to provide a precise requirements definition and then implement exactly those requirements. This results in both a process and end products (software) which are very sensitive to even small changes in requirements. In an architecture-driven process the goal is to produce an architecture that is resilient in the face of changing requirements, within some reasonable bounds. The iterative development process then produces a series of architectural prototypes which result in a robust architecture with the required properties.

Large scale reuse. Object-oriented design and an architecture-driven development process implicitly support reuse. However, field experience has demonstrated that reuse must be an explicit management and technical objective in order to achieve economic results. Reuse is most cost effective when reusing reasonably large components (subsystems or class categories), allowing reuse of the analysis, design, integration, and testing of these larger components. Reusing individual classes or modules is important and effective, but has less leverage than reusing larger subsystems consisting of many pre-integrated classes and prefabricated objects.

Software process control and improvement. The transition to an object-oriented software process introduces new challenges and opportunities for management control of concurrent activities and tangible progress and quality assessment. Real world project experience has shown that a highly integrated environment is necessary to both facilitate and enforce management control of the process. An environment that provides semantic integration (where the environment understands the detailed meaning of the development artifacts) and process automation can improve productivity, improve software quality, and accelerate the adoption of modern techniques. For example, it is difficult to fully exploit iterative development if the turnaround time for system builds is measured in days. An environment that supports incremental compilation, automated system builds, and integrated regression testing can provide rapid turnaround for iterative development and allow development teams to iterate more freely.

Software first focus. The onset of open systems standards (e.g., UNIX, TCP/IP), language standards (e.g., Ada, Ada 9X) with highly portable target implementations (e.g., VADS), distributed architecture middleware (e.g., UNAS) and target platform independent development environments (e.g., Rational Apex) has enabled the selection of target technologies (hardware platforms, operating systems, network protocols, and topologies) to be effectively postponed until the optimal time in a project's life cycle. This is crucial to achieving effective software-based tradeoffs between function, performance, cost and schedule in an environment where target technologies are changing dramatically over a project's life cycle.

Each of these elements is related to the others and the combination of the elements is far more powerful than the individual elements. Implementation of these strategies requires a number of organizational and cultural changes as part of re-engineering the development process. As with other paradigm shifts, one must diverge from many of the accepted management practices towards an improved process which better exploits the strengths of new technologies. Resistance to this change is commonplace, especially since it must originate from the senior ranks of project and organizational leaders who are generally comfortable with the status quo.

U.3.3 Relative Maturity of the Aerospace and Defense Software Practices

In order to compare the state of the practice in the aerospace and defense industry with that of commercial industry we examine the question of how the rate of adoption of modern techniques in the aerospace defense industry compares with the rate of adoption in commercial (non-defense) industries.

While object-oriented techniques have received considerable visibility (some would say "hype") over the last several years, the aerospace industry has actually been a proving ground for many of its concepts as applied to large systems over the last decade. Some of the early successes which demonstrated the economic benefits of object technology occurred in the defense and aerospace industry (primarily using Ada as the implementation language). The 1991 IDC white paper on object technology (targeted at commercial industry) cites the NobelTech (now CelsiusTech) experience as one of the first demonstrations of the economic payoff from moving to object technology. Beginning in 1986 they used object-oriented design, Ada and iterative development to achieve large-scale reuse and significantly enhance their competitive position.

Similarly, TRW and the United States Air Force have extensively documented the successes of architecture-driven development on command and control systems. The Command Center Processing and Display System-Replacement (CCPDS-R) project, the Cobra Dane System Modernization (CDSM) project achieved twofold increases in productivity and quality (primarily reductions in delivered error rates and efficiency of software change) along with on-budget, on-schedule deliveries of large mission-critical systems by employing Ada and an iterative development process substantially similar to that described in the previous section. These improvements were largely due to a major reduction in the software scrap and rework (less than 25%) enabled by architecture-driven iterative development, open-minded acquisition practices, and the use of Ada.

While CelsiusTech and TRW were early adopters, over the last four years we have seen momentum shift toward using modern techniques on most of the large aerospace systems where Rational is involved (admittedly a biased sample, since Rational customers tend to be relatively advanced technologically). This shift in momentum represents a fundamental change from the 1983-1987 time frame when Rational first began to recommend this process to customers in the defense and aerospace industry. At that time most programs used functional decomposition, a waterfall life cycle model, requirements-driven development, etc. There was widespread resistance towards moving to these new techniques on a number of fronts.

Program control. Software managers were concerned that iterative development appeared to turn the programmers loose to start coding without requirements or a design. This violated the traditional standard of the waterfall model (no coding before CDR) and may have been a valid concern at that time, given that iterative development had not been well formalized and documented. Today, Rational and others have successfully demonstrated iterative development and software technologies for rapid prototyping have matured dramatically. It is now well accepted that iterative development actually gives managers greater control over projects than traditional waterfall models.

Military standards. Program managers were concerned that iterative development was inconsistent with DoD-STD-2167 and other military standards. While many would argue that the military standards did not define a development methodology, the reality was that the default interpretation and application of the standards did create significant issues. Over time the standards have become more consistent with modern practices, although many government program offices still interpret the standards in a manner which discourages iterative development and incremental deliveries.

Economics. Some early programs did not see the economic case for reuse. Those companies with a large number of fixed-price contracts in competitive markets and those who were interested in producing a reusable product-line immediately saw the benefits and adapted. Those contractors with large cost-plus contracts who felt secure from competition often saw little economic benefit. The current budgetary environment has begun to change attitudes. Program managers more frequently realize that cost and schedule overruns and poor software quality are likely to result in program cancellations in the current environment, rather than creating additional revenue opportunities. Unfortunately, there are still programs today where the resistance to adopting new technology is not based on skepticism that the technology will provide an adequate ROI, but rather concern that the technology will in fact perform as billed, reducing costs and therefore reducing revenue and profits (again this is primarily an issue for cost-plus or level-of-effort contracts).

Inertia. Most program managers are conservative by nature and do not wish to be early adopters of a new technology. This was certainly a reasonable position to take with respect to Ada, object-oriented design and iterative development in the 1983-1987 time frame. Today the inertia is definitely moving in the right direction toward adopting these techniques throughout the aerospace industry.

These obstacles have been (or are being) overcome and the modern techniques of the object-oriented software process described earlier are becoming increasingly common in the aerospace and defense industry. Many large projects (500,000 lines-of-code or greater) have adopted or are adopting these techniques, and many have experienced very positive results. The actual practice (not just study and evaluation) of this next generation software process in aerospace and defense is as widespread as in any other industry segment. This observation is confirmed both by Rational's direct experience with customers and by all of the survey data available from independent research organizations (which Rational purchases as part of its marketing and business planning activities).

Only in the last three years have we seen general acceptance of object technology, architectural focus, and iterative development in other market segments and the usage there is predominantly exploratory rather than full-scale production. Even in the telecommunications industry, an advanced and sophisticated market, the rate of adoption of new techniques is no faster than in the aerospace industry. Three major factors have contributed to the adoption of modern techniques in the aerospace and defense industry.

Leading edge technology. As with many other technologies (semiconductors, materials, algorithms, etc.), aerospace and defense systems have frequently pushed the limits of software technology because of the scale of the systems being built and the extremely demanding requirements. From distributed systems to massively parallel processing, from enormous databases to extreme real-time performance, aerospace and defense systems continually push the limits of what is possible, while also requiring high reliability and affordability. These pressures have demanded the best possible software engineering technology and motivated the exploration, and then adoption of an object-oriented software process.

Focus on engineering rigor. While some market segments have at times emphasized software development as an art and occasionally encouraged a "hacker" mentality, the aerospace and defense industry has generally viewed software development as fundamentally an issue of engineering discipline. Perhaps because of the deadly serious nature of the defense business, or perhaps because of a similar focus on life-critical software (i.e., commercial avionics and air traffic control systems), the aerospace and defense industry has embraced software engineering as a top priority. This is now true of many other industries (medical instrumentation, telecommunications, etc.) in part because of increasing product liability issues and the focus on total quality management and continuous process improvement. Ironically, this focus on engineering is also largely responsible for producing many of the "classic" methods (functional decomposition, waterfall life cycle model, etc.) which sometimes stand in the way of progress.

Transition to Ada. In the late 1970's and early 1980's when Ada was being developed, the primary focus was on producing a single standard language for embedded and mission critical systems, replacing the 400+ languages in use at that time and thereby reducing the tooling, training, development, and maintenance costs associated with DoD software. While Ada did provide that standard language, an even more important result of the adoption of Ada within DoD has been that Ada has served as a very effective catalyst for the adoption of modern software engineering principles. Some of the early Ada projects did view Ada as "just another programming language" like JOVIAL, Fortran, or C. Those projects basically designed programs the same way they had in previous languages and simply coded them in Ada, achieving few of Ada's benefits while incurring many of the costs of transitioning to a new technology. Fortunately, most of the aerospace and defense industry quickly realized that there was much more to Ada and proceeded to fundamentally re-evaluate all software engineering practices, leading eventually to the adoption of more modern techniques.

On the other side of the ledger, there is one major factor which has inhibited software process improvement in the aerospace and defense industry: the acquisition process.

U.3.4 The Defense Software Acquisition Process

The defense acquisition process and applicable software development standards (e.g., DoD-STD-2167A, MIL-STD-1521B) have historically discouraged the use of iterative development in the defense industry. It is useful to summarize those characteristics of the classic software acquisition process (as it has been typically applied, not necessarily as it was intended) where changes are required in order to enable an object-oriented software process like the one we have described.

Requirements definition. The conventional waterfall model depends upon completely and unambiguously specifying requirements before other development activities, treating all requirements as equally important, and further depends upon those requirements remaining constant over the software development life cycle. These assumptions do not fit the real world. Requirements specification is both the most difficult and the most important part of the software development process. Virtually every major software program suffers from severe difficulties in requirements specification. Moreover, the treatment of all requirements as “equals” has drained massive engineering hours away from the driving requirements and wasted those efforts on MIL-STD-required paperwork associated with traceability, testability, logistics support, etc., which is inevitably discarded later as the driving requirements and subsequent design understanding evolve. The intractability of correctly specifying and prioritizing requirements for complex systems has been one of the primary forces behind the move from the waterfall life cycle model to more iterative life cycle models. Iterative models allow the customer and the developer to work with successive “prototype” versions of the system. Pragmatically, requirements can and must be evolved along with an architecture and an evolving set of application increments so that the customer and the developer have a common understanding of the priorities and an objective understanding of some of the cost, schedule and performance tradeoffs associated with those requirements.

Waterfall architecture and design. Conventional techniques also tend to impose a waterfall model on the architecture and design process which inevitably results in late integration and performance showstoppers. In the conventional model the entire system is designed on paper, then implemented all at once, then integrated. Only at the end of this process was it possible to perform system testing to verify that the fundamental architecture (interfaces and structure) was sound. Iterative development produces the architecture first, allowing integration to occur “as the verification activity” of the design phase and design flaws to be detected and resolved earlier in the life cycle. This replaces the “big bang” integration at the end of a project with continuous integration throughout the project. Iterative development also enables much better quality insight because system characteristics which are largely inherent in the architecture (e.g., performance, fault tolerance, maintainability) are tangible earlier in the process where issues are still correctable without jeopardizing target costs and schedules.

Heterogeneous life cycle format. Given the immature languages and technologies employed in the conventional defense software approach, there was substantial emphasis on perfecting the “software design” prior to committing it to the target programming language where it was subsequently difficult to understand or change. This resulted in the use of multiple formats (requirements in English, preliminary design in flowcharts, detailed design in PDL, and implementations in the target language such as Fortran) and error-prone human-intensive translations between formats. The combination of Ada and iterative development enabled a much more homogeneous representation format across the software life cycle, namely a readable, compilable, and executable library of integrated Ada components which eliminated the need for error-prone translations between different, often incompatible formats, in favor of evolutionary refinements in abstraction and ever-increasing depth and breadth of tangible functionality, quality, and performance. Figure U-1 illustrates the difference in focus between the intermediate products of the two life cycle models.

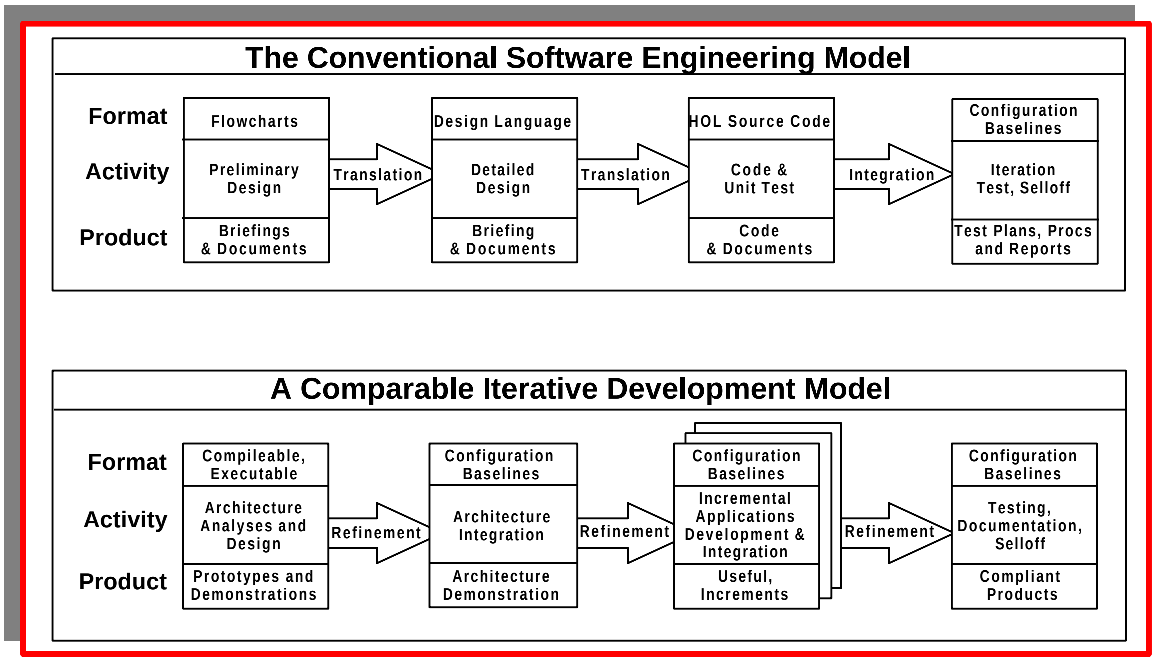


Figure U-1. Iterative Development Products versus Conventional Development Products

Adversarial relationships. In large part because of the difficulties in requirements specification, the conventional process tends to be adversarial, with the customer and the contractor all too frequently locked in mortal combat. Many aspects of the classic acquisition process degenerate into mutual distrust. This makes it very difficult to achieve a balance between requirements, schedule and cost. A more iterative model, with a closer working relationship between customer, user, and contractor, allows tradeoffs to be made based on a more thorough understanding on both sides. This requires a competent and demanding program office with both application and software expertise and a focus on delivering a usable system (rather than blindly enforcing standards and contract terms) and allowing the contractor to make a profit with good performance. At the same time, it requires a contractor who is focused on achieving customer satisfaction and high product quality in a businesslike manner.

Focus on documents. The conventional process has been focused on producing various documents which attempt to describe the software product, with insufficient focus on producing tangible increments of the products themselves. Major milestones are defined solely in terms of specific documents. Contractors are driven to produce literally tons of paper in order to meet milestones (and get progress payments) rather than spending their energy on tasks that would reduce risk and produce quality software. An iterative process requires actual construction of a sequence of progressively more complete systems which (1) demonstrate the architecture, (2) enable objective requirements negotiations, (3) validate the technical approach, and (4) address key risk resolution. Ideally, both the government program office and the contractor would be focused on these “real” milestones with incremental deliveries of useful functionality rather than speculative paper descriptions of the end item vision.

Requirements-driven functional decomposition. A fundamental property of the conventional approach is that it has been very requirements driven with the requirements specified in a functional manner. Built into the classic defense acquisition process is the fundamental assumption that the software itself is decomposed into functional components (CSCIs, CSCs, and CSUs in -2167A terminology), with requirements then allocated to these components. This decomposition is often very different than a decomposition based on object-oriented design and reuse. The functional CSCI decomposition becomes anchored in contracts and subcontracts, often precluding a more architecture-driven approach.

NIH (not-invented-here). The conventional process can discourage reuse between projects and tends to discourage the use of commercial technology. Since requirements are often "thrown over the wall" to the software developer, there is little opportunity to negotiate the compromises that are required to reuse an existing product or subsystem. Furthermore, effectively building reusable components or subsystems necessitates investment above and beyond that required for the narrow scope of the project at hand. Ideally the process would encourage customers and contractors to invest in developing reusable architectures which could be applied to a variety of systems in a given domain (avionics, C3I, etc.). Instead, the current process and incentives prevent most investment in reuse through encouragement of specific and singular contract-selfish performance.

Economic incentives. As part of the adversarial nature of the acquisition process, there is considerable focus on ensuring that contractor profits are within a certain acceptable range (typically 5-15%). Occasionally, excellent contractor performance, good value engineering, or significant reuse result in potential contractor profit margins in excess of "their acceptable initial bid." As soon as customers (or their users or government SETA organizations) become aware of such a trend, there is inevitably substantial pressure applied to employ these "excess" resources on out-of-scope changes until the margin is back in the acceptable range. As a consequence, the simple profit motive which underlies commercial transactions and incentivizes efficiency is replaced by complex contractual incentives (and producer-consumer conflicts) which are usually suboptimal. Very frequently, contractors see no economic incentive to implement major cost savings, and certainly there is little incentive to take risks which may have a large return. On the other side of the ledger, contractors can easily manage to consume large amounts of money (usually at a small profit margin) without producing results and with very little real accountability for poor performance.

The success of new technologies has led to a more widespread view that the classic defense software acquisition process must be modified or replaced. The new MIL-STD-498, replacing DoD-STD-2167A and DoD-STD-7935A, represents a partial recognition of this problem. The goals of MIL-STD-SDD include removing the implied waterfall model, removing the implied preference for functional decomposition, providing clearer requirements for software reuse, and lessening the emphasis on documents. However, very few of the issues expressed above are dealt with in an explicit manner in the new standard and experience to date has indicated that it is still very difficult to implement iterative development, since most program offices do not understand the new technologies. Even where there is a relatively advanced program office in favor of using modern practices, the various matrix entities (e.g., IV&V contractors, FFRDC, and SETA contractors, etc.) are wedded to the old process model and are more concerned with protecting their turf (and their jobs) than with producing systems in a more cost effective manner.

U.3.5 ADA and the Aerospace and Defense Industry

Ada is an outgrowth of a remarkable vision that was first enunciated over fifteen years ago. Today, Ada is almost universally recognized as the software industry's premiere language for mission critical software engineering. The struggle to transform the Ada vision into reality (via very useful products) has been pursued with surprising intellectual vigor even though it was far from being the most popular language initiative of the computer science community. Ada's principle raison d'être is the DoD's need for a single language in which the software engineering paradigm was supported by, and in some instances enforced within, the semantics of the language. This objective has been very nearly achieved. The table below identifies how the DoD contractor community viewed Ada's risks in 1985 versus risk resolution focus emphasized today. The evolution depicted below represents remarkable progress which is a tribute to DoD and the Ada community.

1985 RISK RESOLUTION FOCUS	1994 RISK RESOLUTION FOCUS
Compiler availability and maturity	Development resource adequacy
Ada language training	O-O and architecture training
Ada software development costs	Reuse costs
Ada environment tool availability	Tool integration and extent of automation
Ada process definition	Iterative development process improvement
Ada/COTS interfaces	Open systems interoperability
Ada runtime overhead	Target resource adequacy

Table U-1 Ada Risk Evolution from 1985 to 1994

Ada 83's semantics can be characterized as providing strong support for project management functions; somewhat lesser support is provided to the advanced computer science attributes of object-oriented programming which evolved after Ada 83 was baselined. These drawbacks however, will be substantially corrected by Ada 9X. In spite of Ada's success, the invention of new languages has continued unabated in commercial industry. C++ is one example of a relatively new language whose primary design goal was to provide object-oriented programming support (encapsulation, abstraction, polymorphism and inheritance) without compromising the advantages of C (primarily speed and ease of programming). In contrast to Ada, C++ provides little project management support in its selected semantics but it is designed for stronger support to the computer science attributes underlying object-oriented design.

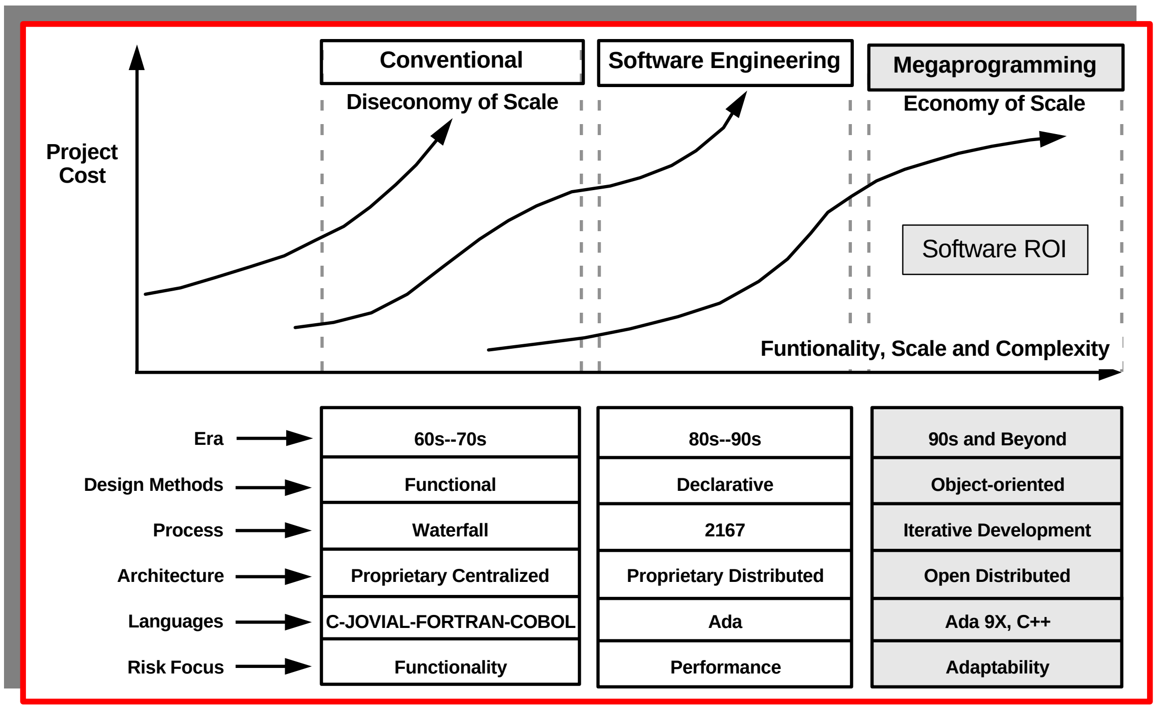
The definition of the Ada language is unique in that it was designed with the goal of enabling better management, design, and architectural control (the higher leverage aspects of software engineering) while sacrificing some of the ease of programming. This is the essence of the Ada culture: top-down control where programmers are subordinates of the lead architects and managers. Other languages, and specifically C++, are focused at simplifying the programming activities while sacrificing some of the ease of control. This of course, is the essence of the C/C++ culture where programmers lead the way. For small programs and noncritical projects, the C++ culture can work well and the Ada culture is perhaps overkill. But for large, complex mission critical systems, the Ada culture is a field-proven necessity for success. Culture is a human-imposed set of trends. Clearly, an Ada culture can be practiced with C++ and vice versa, but the paradigm shift for an organization with cultural inertia is an emotional and extremely difficult undertaking.

It is interesting to note that in the definition of the C++ language, there are many new features which were clearly influenced by earlier Ada advances. Similarly, the O-O features being incorporated in Ada 9X have clearly been motivated by advances in C++. The point here is that both languages have contributed to each other's technical evolution. This language competition has been healthy and while there are numerous rhetorical debates about which of Ada or C++ is better, there is very little debate that both of these languages are a quantum leap above all others in supporting the modern techniques of object-oriented software engineering as described within this paper.

As indicated earlier, the single greatest contribution of Ada was to act as a catalyst for the adoption of modern software engineering practices. There has been substantial progress in the software technology in use within this industry over the past decade. Figure U-2 depicts the on-going transition of software economics from the conventional “dis-economy of scale” (caused by the dominance of custom development, ad hoc processes and ad hoc environments) to the emerging “megaprogramming” economy of scale being achieved by organizations who exploit reuse, integrated environments with high levels of automation, and mature, iterative development techniques. Ada, and its associated improvements in environments and process, was to a significant extent, the intermediate catalyst in this transition. In many ways it has been Ada which has turned software engineering into a true professional engineering discipline within this community. Ada provided a truly standard and portable language, widely available on virtually all hardware platforms, with extensive support for modern software engineering principles. Ada has also been a vehicle for introducing new life cycle models, new tools, new design and programming practices, and more secure approaches to the development of high-reliability and safety-critical software. Considerable momentum has been established and this momentum is accelerating with the recent emphasis (in both Ada 83 and Ada 9X) on the use of object-oriented analysis and design with Ada.

Recently there has been considerable discussion of the DoD policy toward Ada, with some polarization between those who believe the current policy should be continued or strengthened and those who believe that the current policy should be abandoned. It is not necessary here to repeat all of the arguments pro and con, but it is useful to examine the key positions and assess their validity from the perspective of Rational’s experience. The arguments for continuing the Ada mandate can be reduced to the following:

Technical. Virtually every language evaluation study we know of has concluded that Ada is the best technical language for the DoD domain. Ada has satisfied the goals of the DoD in being a highly reliable and maintainable language. Its strengths include support for large scale projects, ultra-reliable software development, standardization, and real-time support, exactly the needs of the defense domain and other mission critical domains where complexity control and certifiability are required. Ada mandate risks have been substantially resolved whereas the other leading alternative (C++) is faced with many of the same risks that Ada faced 10 years ago (see Table U-1).



Era	60s--70s	80s--90s	90s and Beyond
Design Methods	Functional	Declarative	Object-oriented
Process	Waterfall	2167	Iterative Development
Architecture	Proprietary Centralized	Proprietary Distributed	Open Distributed
Languages	C-JOVIAL-FORTRAN-COBOL	Ada	Ada 9X, C++
Risk Focus	Functionality	Performance	Adaptability

Figure U-2 Progress Towards Improved Software Economics

Inertia. The past 10 years of DoD investment have resulted in a substantial base of Ada assets including compilers, training, reusable components, and case studies. Furthermore, despite the mandate, Ada is commonly selected by DoD contractor preference and there are several domains (global air traffic control, NASA, Nuclear Power, FAA, NATO, etc.) that employ Ada in the absence of any mandate.

Standardization. DoD's business case is very different than commercial industry's. The need for a standard language in DoD is motivated by their current practice of organic maintenance with high personnel turnover rates, whereby the costs of tooling and training their maintenance force for a single language have huge economic benefits. This was, in fact, the dominating requirement for Ada's development: to eliminate the divergence of languages, support environments and lack of any ROI in personnel training from assignment to assignment. This need is certainly just as important today as it ever was.

Economic. A substantial number of very large applications (greater than one million source lines) have been successfully delivered and maintained in Ada. While there is certainly not universal success in the financial performance of Ada projects, there is substantial evidence that a mature software organization will perform better with Ada than other languages. There are two important trends of note:

Prior to Ada, there were (close to) zero large scale projects that delivered on-budget or on-schedule. Over the last 10 years of employing Ada there have been several well-publicized successes.

Across all projects that have been "less than successful," we know of none that attributed their failure in whole, or in part, to Ada.

The arguments for eliminating the Ada mandate can be summarized as follows:

No object-oriented support. Ada 83, while clearly not supportive of all the object-oriented programming features in vogue today, does support many of the techniques and processes of object-oriented software engineering as described earlier in this paper. In fact, most of the large-scale successful Ada projects Rational is familiar with, were successful predominantly because they were employing object-oriented techniques and modern processes. The primary leverage of these modern techniques is in the process and architecture focus, not in the programming language support. Furthermore, the Ada community has embraced the advantages of object-oriented programming support directly in the language as evidenced by their inclusion in Ada 9X.

Lack of commercial support. The argument that Ada lacks support in the commercial marketplace is subtle. On one hand, anyone that walks into any bookstore's software section will find over 20 books on C++ and maybe 1 on Ada. On the other hand, despite their scarcity in commercial bookstores, there are 30+ textbooks on Ada and it is becoming an increasingly popular vehicle for teaching software engineering at universities. There are numerous non-DoD projects who employ Ada for technical and financial reasons. In general, these commercial applications are similar to DoD applications in scale and complexity and the organizations chose Ada for the same reasons as DoD. Perhaps there would be more acceptance of Ada in other commercial applications if DoD had done a better job of marketing, but then again, perhaps not. The real issue here is whether DoD and commercial domain must be closely in synch. A large percentage of the commercial market's software is totally incongruent with DoD software and many commercial practices are equally inappropriate to most DoD software (the glaring exception is DoD's MIS systems which only differ by perhaps their scale). The principal inhibitor of Ada's commercial perception is probably the lack of PC based tools. The impact of PCs on training, software development, and available COTS products is profound. The huge installed base of PCs drives the software market trends and the lack of Ada support on PCs inhibits the single largest source of cheap computing cycles from being part of the Ada solution space.

The GNU Ada Translator will help this problem considerably and the emerging next generation PC operating systems will enable today's Ada environments to be more easily transitioned to PC platforms.

Insignificant Ada market segment. This argument is closely related to the previous one but rather than focusing on commercial projects, we examine commercial product markets. In 1992, the Ada compiler and tool market was somewhere in the range of $200-300 million while the C++ compiler and tool market was $300-500 million. While the Ada market is certainly smaller, it is by no means insignificant, and there is ample demand to stimulate considerable investment by small and large companies.

Conflict with use of COTS. There is little debate that there are fewer COTS products available for Ada than there are for some other languages. However, we see no real issue with integrating Ada with COTS products. Most of the globally important interfaces (DBMSs, GUIs, operating systems, network protocols) have been worked. Furthermore, there are Ada-based COTS products for development environments (e.g., Apex, VADS), and architecture middleware (e.g., UNAS) which are better than other language counterparts and provide proven leverage in achieving technical and financial success in large complex software projects.

As implied above, we believe that DoD should stand firm on the Ada mandate. In parallel, however, we support the continuing development of both Ada and C++ and DoD should support Ada-C++ interoperability advances which will continue as the C++ language matures (particularly with respect to compiler integrity and language standardization and control) to the current levels of Ada and Ada 95. Ada vendors are already investing aggressively to support this interoperability, C++ vendors should be equally as open. DoD should not consider dropping the Ada mandate; it is an asset in DoD business model which should not be polluted by the "in vogue" trends towards commercial practices. Re-evaluating this position and perhaps opening up the mandate to both Ada and C++ after C++ matures into a standard makes good business sense given the rate of software technology advance. However, this maturity level is not likely to occur before 1998. We believe that this long term strategy would promote further investment in Ada-C++ interoperability (which is good for both commercial and DoD domains) and permit some level of healthy competition to continue.

U.4 Recommendations

There are several general recommendations that Rational would suggest to policy makers and industry leaders in defense and aerospace. Rational recommends that DoD (and other agencies such as the FAA, NASA, etc.) be more demanding customers with a focus on results. DoD, while not so dominant that it drives the entire software industry, is still a large customer with significant clout. By demanding quality software at reasonable prices on reasonable schedules, DoD can, and will, impact the behavior of the industry. Demanding performance and refusing to tolerate failure will strengthen the industry and encourage the adoption of best practices. Programs with a significant track record of poor management, severe cost and schedule overruns, and poor software quality (defined in terms of fitness for use, not just defects and compliance with narrow specifications) should be terminated promptly, regardless of fault (Government, contractor, whatever). This must be tempered with the understanding that software development contains risk and one must not discourage appropriate risk taking. However, over the medium- and long-term a more businesslike and demanding attitude on the part of the Government (similar to the commercial market where weak products and producers are eliminated rapidly upon evidence of failure) will be much less expensive than continuing to subsidize poor performance. Without doubt, this is the single most important recommendation we can make.

Rational recommends full and continued support for Ada as a centerpiece of aerospace and defense software policy (DoD, NASA, FAA). As discussed above, we believe there are strong technical and business reasons for using Ada in defense applications.

Rational recommends that DoD continue to encourage the use of commercial-off-the-shelf technology. Procurement and project management practices must consistently encourage use of commercial technology. Today there is insufficient incentive to use commercial technologies and there is absolutely no incentive to compromise often arbitrary requirements in order to allow the use of commercial technology. While some may disagree, we view this as synergistic with the Ada initiative.

Rational recommends continued efforts to streamline and modernize the software acquisition process. The new MIL-STD-498, replacing DoD-STD-2167A and DoD-STD-7935A, is a step in the right direction but does not go far enough with respect to the state-of-the-practice. The pace of such a global change remains excruciatingly slow and most program offices do not understand modern software engineering principles well enough to properly manage software acquisition with the new standard. Further promotion and adoption of iterative development processes (where success and failure signals are more obvious and tangible earlier in the life cycle) is also critical to achieving any kind of success towards our first recommendation. DoD (and the contractor community) must institute a more aggressive program of process improvement to more rapidly evolve the defense software acquisition process into a quality process. DoD must become less insular, reaching out to understand the best practices and lessons-learned in other software markets (more specific recommendations are included below).

Rational recommends stronger support for applied research in software technology in the US Traditionally, software-related technology efforts have been extremely under-invested by both the government and defense contractors. For example, recent awards for the Technology Reinvestment Program were quite discriminatory against funding software related efforts despite the rapidly growing importance of such technologies. Software technology remains a "core competency" of US industry. Not only is advanced software technology developed within the US, but it is rapidly exploited and applied (unlike some other technologies). The US software market remains the largest and the most competitive worldwide and all of the participants (including the defense and aerospace industry) benefit from the dynamic nature of this market. Further investment would maintain this commercial competitiveness as well as benefit DoD software marketplace.

Rational recommends that DoD institute a required training program for all DoD project offices involved in acquisitions with software content greater than some threshold (say $1-5M). This program should be modeled after the Air Force's BOLDSTROKE course but contain more up-to-date project case studies and more focus on software project management and acquisition. Furthermore, while DoD has successfully applied the SEI's Software Capability Evaluations to discriminate contractors with software process maturity, it has yet to apply similar discipline to its own acquisition project offices.

U.5 About the Authors

Mike Devlin cofounded Rational in 1981 and served as a member of the Board, Executive Vice President and Chief Technical Officer until he was elected Chairman of the Board in December 1989. Mr. Devlin was appointed Chairman of the Board of Rational Software Corporation and formed from the combination of Rational and Verdix Corporation in March 1994. Mr. Devlin is a graduate of the United States Air Force Academy and was associated with the Air Force Space Division and Satellite Control Facility as a software program manager and as a computer scientist. Mr. Devlin was the Space Division's liaison to the Defense Advanced Research Project Agency on issues relating to modern software languages and methodology. Mr. Devlin graduated first in the class of 1977 at the Academy and was the outstanding graduate in each of his two major fields of study, Engineering and Computer Science. He was awarded a National Science Foundation Graduate Fellowship and received a MS degree in Computer Science from Stanford University in 1978.

Walker Royce is the Director of Software Engineering Process for Rational Software Corporation. Prior to joining Rational Software Corporation, Mr. Royce spent 16 years in a variety of software technology and software management roles at TRW. He was the Project Manager of the Universal Network Architecture Services (UNAS) product-line where he defined and managed its state-of-the-art software process. He served as the Software Chief Engineer responsible for the software process, the foundation Ada components and the software architecture on the CCPDS-R Project, a highly successful, million-line Ada project. Mr. Royce led the development of TRW's Ada Process Model and the UNAS product technologies which have been transitioned from research into practice on numerous large projects and earned him a TRW Technical fellowship and TRW's Chairman's Award for Innovation. His pioneering work in advancing distributed software architecture and evolutionary software process technologies have been published in numerous technical articles and guidebooks and he is a featured lecturer at the Air Force BOLDSTROKE forum on Software Management. Mr. Royce received his B.A. in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and completed three years of further postgraduate study in Computer Science at UCLA.